Question 1: An Authentication Program

Checkpoint 0. Investigate the Login program and login as one of the users. State the correct arguments that are accepted by the Login program.

The command form is .\Login <Username> <Password>. If you examine the database you can find the user names along with hashed passwords. You can try to break one of the passwords with a dictionary attack, or make an educated guess such as:

./Login user user

Checkpoint 1. What does the output of findbugs tell you?

FindBugs tell you that there is a possible SQL injection (CWE-89) because user input is used directly to create SQL statement by string concatenation. FindBugs also tells you that the statement is not closed after use.

Checkpoint 2. Which part of the authentication program is vulnerable to SQL injection and how can an attacker exploit it?

In the method doPrivilegedAction, the user parameter is used directly for the creation of SQL statement by string concatenation without any escaping or checking process. An attacker can add arbitrary SQL code to corrupt the database or affect the retrieved result.

Checkpoint 3. Why isn’t the removal of quotes and semicolons through sed adequate to protect against SQL injection?

Firstly it doesn’t remove all SQL special character like comments or wildcard characters. More importantly there is an echo -e after the sed command which will interpret all ‘\’ escaped hex characters which may expand to quote and semicolons characters as their ASCII values.

Checkpoint 4. Give your patch to repair the Login and Login.java programs and explain how it works.

Use absolute path for sed command. Avoid echo -e command after sed operation. Use a prepared statement in Java for database query preparation.

Checkpoint 5. Examine the exploits and explain how they work. What design flaws or vulnerabilities are abused by the exploits?

GPG Password: time of check to time of use

Both of them try to shorten the SQL query. One of them provides the hex value representation of quote and semicolons which will be interpreted by the echo command. This is the problem of “time of check to time of use”. The other one lets the Linux system search for the sed binary in a self-defined directory, which allows a malicious sed to be executed. This is abuse of the PATH variable as linux will search for unknown executable binary in the order of directory listing provided by PATH variable.

Checkpoint 6. Did your patch stop the exploits? If not explain why and provide an updated patch that does.
Question 2: Another SQL injection

Checkpoint 0. What are the users already in the database and what attacks against them can you imagine?

The users are admin and user. A second order SQL injection attack is possible. This allows an attacker to change the password for any user illegally.

Checkpoint 1. When you run the findbugs program on the class file it shows there is an SQL injection problem despite the use of a prepared statement. Why?

Even though the username has been saved to the database with a prepared statement, we don’t check it for SQL metacharacters. Consequently whilst we cannot do any exploits while logging in or signup, we can do some tricks while changing the password.

Checkpoint 2. Fix the code and provide a patch for Login.java.

Avoid naked string concatenation on creating sql query. Use prepared statement instead. Malicious user input may not have immediate effect, it may work if it is reused in some later operation. It is called as second order SQL injection. So it is better to check for and deny SQL metacharacters before storing into the db.

Checkpoint 3. Describe how the exploit works.

GPG Password: little bobby tables

It is a second order SQL injection. If you create a user with a silly username like admin’;– You can perform bad operation later to mess up the query logic and overwrite an admin password.

Checkpoint 4. Verify your patch stops the exploit. If not, make a new patch that does.

Question 3: Vulnerable Voting App

Checkpoint 1. What is the difference between a first order SQL Injection attack and a second order SQL Injection attack?

First order SQL Injection work directly when the attacker first submit the malicious input. Second order SQL Injection normally exists because of incomplete protection. Sometimes the malicious input has been sanitize when it is first inputted by the attackers and stored in somewhere, but when these information is reused later on, the sanitization or checking is incomplete or missing that allows the stored problematic value because an attack later on. The best way is to add in sanitization any time or even block (but not escape) problematic input at the first moment.

Checkpoint 2. Briefly describe the steps you need to login as a legitimate user. You may want to examine the function signup() and function login() in /srv/http/include/functions.php

Although the signup function has escape all special characters in the user name and password and make the attack not working at the begining. But the login function is careless to not include prepare statement and the stored escaped character can still in play in the login script. If a user create a user account with user name arthur’–, it will success. But when the user login with this newly created account arthur’–, it will login to the user account of arthur instead because of the
missing of checking mechanism in the login function. Thus the attacker can use this method to login to any user account without knowing their password.

**Checkpoint 3.** What is an CSRF attack? What relationship is an CSRF attack abusing?

CSRF attack is trying to trick a legitimate user to send a request on their behave (identity) without they even known about it. It is abusing the trust from the server to the user. The server believes that every request from client side linked with an legitimate identity is always a willing action of that user. CSRF try to abuse this relationship.

**Checkpoint 4.** What is the different between the two kind of CSRF attacks. Which one is more dangerous?

Stored CSRF normally inject the request in some location, like img tag or iframe where the user browser will always automatic fetch (call the url) for those resources and sending request accordingly. While redirect CSRF normally ask a user to click on some link or attracked the user to visit some interesting page which contains the request that the attacker wants the victim to send. Stored CSRF is generally more dangerous because with the support of img tag or iframe, it is possible to successfully crafted a CSRF attack without additional action from the victim at all. The only requirement is that user is already login on the specific website which the request is needed.

**Checkpoint 5.** If the same origin policy has been activated, are both type of CSRF attacks still possible in this vulnerable apps?

Same origin policy can only stopped some of the CSRF attack. It can stop CSRF attack which request user click on a link from a different base url (and it send a request to the needed service), but if the web app itself is vulnerable (like our voting apps), t is possible to store the CSRF attack within the same site and thus same origina policy still cannot stop that attack.

**Checkpoint 6.** Briefly describe the steps needed for preparing a stored CSRF. Describe your steps in both the attacker’s and victim’s perspective. You may want to examine the voting logic and how the listing of all user’s image is done.

In this voting app, each user can submit an image link to join in the competition. And the web app naively display all link without checking if it is really an image. So an attacker can actually submit a request to vote for himself as an image link. In this case, every legitimate user will fetch that fake image link and vote for the attacker transparently when they login and see the listing of image. Furthermore, the attacker can also carefully crafted a payload like `<img src="vote.php?vote=1` to both display an legal image and sneak in a voting request.

**Checkpoint 7.** What is the potential problems for allowing the directory listing to be displayed?

The attacker can understand the structure of the webapps and possible attacking the web server by visiting some sensitive files. If poor configuration is exists, it may also allows the attacker to access folders in the local web server which is outside of the web apps base directory.

**Checkpoint 8.** Try access different directory listing of the web app. Are there any serious security concern?

Yes, we can access /db folder which list out our db file and an attacker can download the db and know all the user data.
Checkpoint 9. What is the information displaying in this page?

This is all the settings of the web server and php client.

Checkpoint 10. Why this page should not be accessible in the production environment when the web app is putting into real use? What is the potential problem of showing these information to the public user (or potential attackers)?

Gathering information is always an essential part for an attacker. There are many version and platform specific attacks announce all over the internet. The more the attacker know about the server, the more an attacker can find ways to attack the system. These phpinfo even provide the attacker information about which service or configuration is turned on. Thus make the life and work of an attacker very easy.

Checkpoint 11. (Very Optional) There are far more poor configurations and security problems in the settings and code in this web app. If you have more time, you may try to list them out and discuss how you may fix it.

Example list of poor configurations and security problems

- Directory listing is on (Information Gathering)
- Function phpinfo is printed in administrator.php (Information Gathering)
- No checking of user input link (Injection / Second Order Attack / XSS in old version)
- No checking and confirmation of GET request action (CSRF)
- Warning message and debugging mode is on (Information Gathering)
- No escape of special characters (Injection / Second Order Attack)
- Cookie value is too easy to replicate (Session Fixation / Session Hijacking) (Not encrypted or just hash of username)
- Allow download of php file (Source code viewing)
- Allow download of db file (Database leakage)
- Database contains plaintext password / Database not encrypted (Database leakage)
- Bad DB error handling (No rollback handling, may cause data not sync) (Data Async / TOCTTOU)
- allow_url_fopen is on, possible code injection (Injection)

Question 4: Linkers

Checkpoint 1. What program is used to interpret the ELF file Vulnerable?

/lib/ld-linux.so.2

Checkpoint 2. What is an untrusted search path? How it related to dynamically linking of library?

The binary tries search for some resources or libraries from an externally provided location. This is named as untrusted search path. Dynamic linking of libraries make the binary to find some of the resources or needed library classes at run time, which provide a chance for an attacker to direct the binary to a malicious or modified resources or library classes using untrusted search path technique which the binary have no control or maybe not aware of the action.
Checkpoint 3. How could the **Vulnerable** program be fixed to avoid the search path exploit?

Use static linking of libraries. There is more suggestions on CWE-426 Untrusted Search Path documentation.

Checkpoint 4. Describe what each of the provided exploits does, how likely you consider it to be an achievable exploit and under what circumstances.

One cracks the password... given the complexity of the password it is unlikely to be cracked quickly, however this is still a threat as SHA1 isn’t the greatest hashing algorithm for passwords. Also, attacker may look for carelessly stored password or hash by reverse engineering of the binary itself.

One uses preloading... very likely given the implementation, however if the program ran as a setuid program *most* modern linkers would ignore it.

The last modifies the binary and removes the conditional jump based on the output of the `memcmp`. Extremely likely (this was a common way of cracking software).

Checkpoint 5. How could you prevent each of the attacks?

Password Cracking Defend by obfuscating the password, and considering a different authentication scheme.

Preloading Defend by statically linking.

Binary Patching Near impossible to stop. Solutions can include using specialist hardware (i.e. ROM with some hardware protection) to store and run the program, or using a cryptographic packer to provide obfuscation.

Checkpoint 6. (optional) Modify `exploit-ld.sh` so it attacks the SHA1 call. Similar to exploit-ld.sh Provide a modified version of SHA1 call which always copy the needed hash value to the dest address provided in the argument. Use `memcpy` instead of `strcpy` or `strncpy` to avoid problems caused by zero byte in the hash.